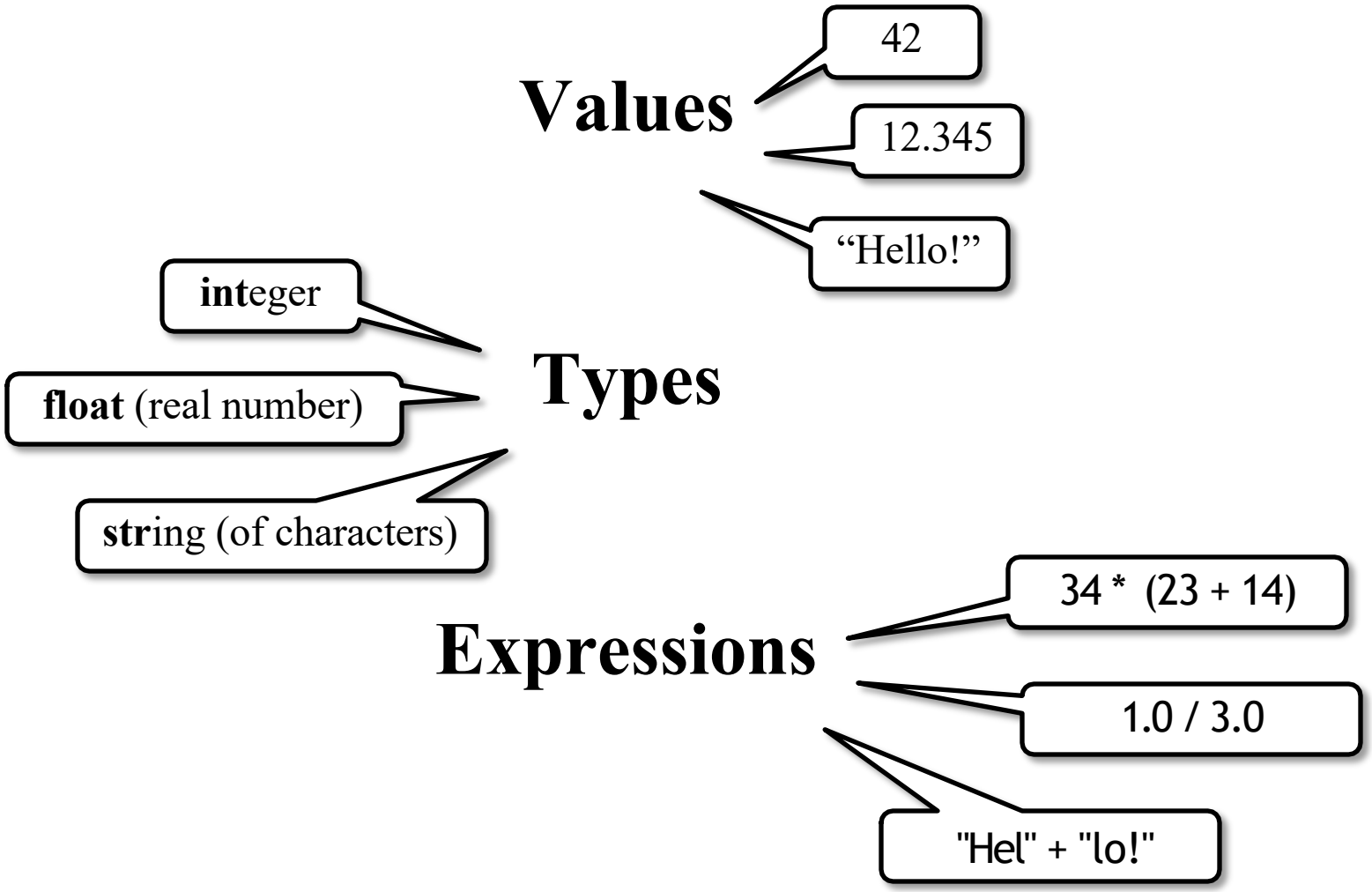
Lecture 02

Python Basics

Why Programming in Python?

- Python is **easier for beginners**
  - A lot less to learn before you start "doing"
  - Designed with "rapid prototyping" in mind
- Python is **more relevant to non-CS majors**
  - NumPy and SciPy heavily used by scientists
- Python is a more **modern language**
  - Popular for web applications (e.g. Facebook apps)
  - Also applicable to mobile app development

The Basics


Values
42
12.345
“Hello!”
Types
integer
float (real number)
string (of characters)
Expressions
34 * (23 + 14)
1.0 / 3.0
"Hel" + "lo!"

Expressions and Values

- An **expression** represents something
  - Python *evaluates it,* turning it into a **value**
  - Similar to what a calculator does
- Examples:

```
>>> 2.2
2.2
>>> (3 * 7 + 1) * 0.1
2.2
```

Expression (Literal)

Value

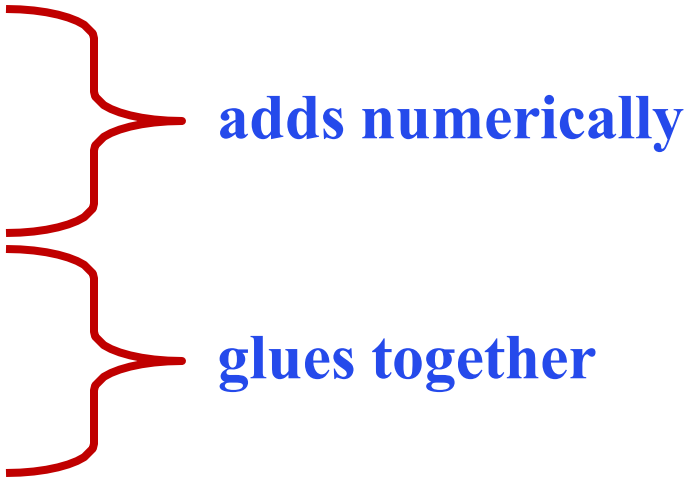
Expression (Complex)

Value

What Are Types?

- Think about + in Python:

```
>>> 1+2
3
>>> "Hello"+"World"
"HelloWorld"
```



- Why does + given different answers?
  - + is different on data of different ***types***
  - This idea is fundamental to programming

What Are Types?

A **type** is both

- a set of *values*, and
- the *operations* on them

Example: int

- **Values:** integers
  - …, –1, 0, 1, …
  - Literals are just digits:
    1, 45, 43028030
  - No commas or periods
- **Operations:** math!
  - +, −(add, subtract)
  - *, // (mult, divide)
  - ** (power-of)

Example: int

- **Values:** integers
  - …, –1, 0, 1, …
  - Literals are just digits: 1, 45, 43028030
  - No commas or periods
- **Operations:** math!
  - +, −(add, subtract)
  - *, // (mult, divide)
  - ** (power-of)

- **Important Rule:**
  - **int** ops make **ints**
  - (if making numbers)
- What about division?
  - 1 // 2 rounds to 0
  - / is **not** an **int** op
- Companion op: %
  - Gives the remainder
  - 7 % 3 evaluates to 1

Example: float

- **Values:** real numbers
  - 2.51, -0.56, 3.14159
  - Must have decimal
  - 2 is **int**, 2.0 is **float**
- **Operations:** math!
  - +, −(add, subtract)
  - *, / (mult, divide)
  - ** (power-of)
- Ops similar to **int**
- **Division** is different
  - Notice /, not //
  - 1.0/2.0 evals to 0.5
- But includes //, %
  - 5.4//2.2 evals to 2.0
  - 5.4 % 2.2 evals to 1.0
- Superset of **int**?

float values Have Finite Precision

- Try this example:

  >>> 0.1+0.2

  0.30000000000000004

- The problem is **representation error**
  - Not all fractions can be **represented** as (finite) decimals
  - **Example**: calculators represent 2/3 as 0.666667
- Python does not use decimals
  - It uses IEEE 754 standard (beyond scope of course)
  - Not all decimals can be **represented** in this standard
  - So Python picks something close enough

int versus float

- This is why Python has two number types
  - **int** is **limited**, but the answers are always **exact**
  - **float** is **flexible**, but answers are **approximate**
- Errors in float expressions can propagate
  - Each operation adds more and more error
  - Small enough not to matter day-to-day
  - But important in scientific or graphics apps (high precision is necessary)
  - Must think in terms of **significant digits**

Using Big float Numbers

- **Exponent notation** is useful for large (or small) values
  - –22.51e6 is –22.51 * 10^{6} or –22510000
  - 22.51e–6 is 22.51 * 10^{-6} or 0.00002251

A second kind of **float** literal

- Python *prefers* this in some cases

```
>>> 0.00000000001
1e-11
```

Remember: values look like **literals**

Example: bool

- **Values:** True, False
  - That is it.
  - Must be capitalized!
- **Three Operations**
  - b and c
    (True if **both** True)
  - b or c
    (True if **at least one** is)
  - not b
    (True if b is **not**)

- Made by comparisons
  - **int**, **float** operations
  - But produce a **bool**
- Order comparisons:
  - i < j, i <= j
  - i >= j, i > j
- Equality, inequality:
  - i == j (**not** =)
  - i != j

Example: **str**

- **Values:** text, or *sequence of characters*
  - String literals must be in quotes
  - Double quotes: "Hello World!", " abcex3$g<&"
  - Single quotes: 'Hello World!', ' abcex3$g<&'
- **Operation:** + (catenation, or concatenation)
  - 'ab' + 'cd' evaluates to 'abcd'
  - concatenation can only apply to strings
  - 'ab' + 2 produces an **error**

Variables & Assignments

Type: Set of values and the operations on them

- Type **int**:
  - **Values**: integers
  - **Ops**: +, –, *, //, %, **
- Type **float**:
  - **Values**: real numbers
  - **Ops**: +, –, *, /, **
- Type **bool**:
  - **Values**: **True** and **False**
  - **Ops**: not, and, or

- Type **str**:
  - **Values**: string literals
    - Double quotes: "abc"
    - Single quotes: 'abc'
  - **Ops**: + (concatenation)

Will see more types in a few weeks

Example: `str`

- **Values:** text, or *sequence of characters*
  - String literals must be in quotes
  - Double quotes: "Hello World!", " abcex3$g<&"
  - Single quotes: 'Hello World!', ' abcex3$g<&'
- **Operation:** + (catenation, or concatenation)
  - 'ab' + 'cd' evaluates to 'abcd'
  - concatenation can only apply to strings
  - 'ab' + 2 produces an **error**

Converting Values Between Types

- Basic form: *type*(*expression*)
  - This is an expression
  - Evaluates to value, converted to new type
  - This is sometimes called **casting**
- **Examples:**
  - float(2) evaluates to 2.0 (a **float**)
  - int(2.6) evaluates to 2 (an **int**)
  - Note information loss in 2nd example

Converting Values Between Types

- Conversion is measured *narrow* to *wide*

$$\textbf{bool} \Rightarrow \textbf{int} \Rightarrow \textbf{float}$$

- **Widening:** Convert to a wider type
  - Python does automatically
  - **Example:** 1/2.0 evaluates to 0.5
- **Narrowing:** Convert to a narrower type
  - Python never does automatically
  - **Example:** float(int(2.6)) evaluates to 2.0

Operator Precedence

- What is the difference between these two?
  - 2*(1+3)
  - 2*1 + 3

Operator Precedence

- What is the difference between these two?
  - 2*(1+3) **add, then multiply**
  - 2*1 + 3 **multiply, then add**
- Operations are performed in a **set order**
  - Parentheses make the order explicit
  - What happens when no parentheses?

Operator Precedence

- What is the difference between these two?
  - 2*(1+3) **add, then multiply**
  - 2*1 + 3 **multiply, then add**
- O
  - **Operator Precedence**:
  - The *fixed* order Python processes operators in *absence* of parentheses

Precedence of Python Operators

- **Exponentiation**: **
- **Unary operators**: + –
- **Binary arithmetic**: * / %
- **Binary arithmetic**: + –
- **Comparisons**: < > <= >=
- **Equality relations**: == !=
- **Logical not**
- **Logical and**
- **Logical or**

- Precedence goes downwards
  - Parentheses highest
  - Logical ops lowest
- Same line = same precedence
  - Read "ties" left to right
  - Example: 1/2*3 is (1/2)*3

Expressions vs Statements

Expression

- **Represents** something
  - Python *evaluates it*
  - End result is a value
- Examples:
  - 2.3 (Literal)
  - (3+5)/4 (Complex)

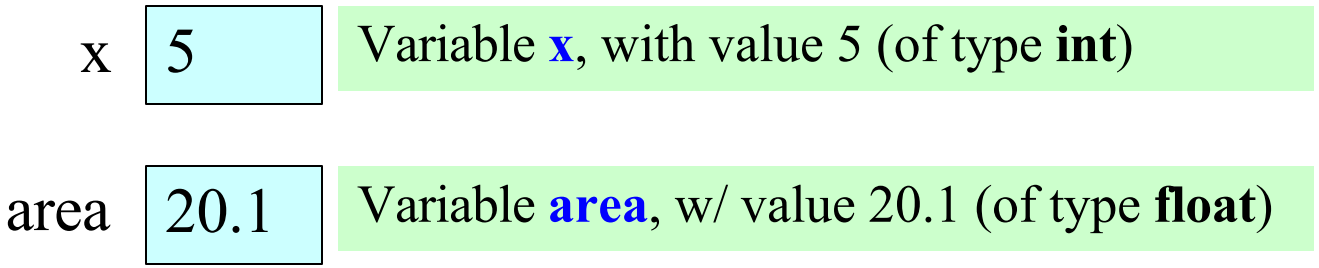
Statement

- **Does** something
  - Python *executes it*
  - Need not result in a value
- Examples:
  - print('Hello')
  - import sys

Will see later this is not a clear cut separation

Variables

- A **variable**
  - is a **box** (memory location)
  - with a **name**
  - and a **value** in the box
- Examples:



Using Variables

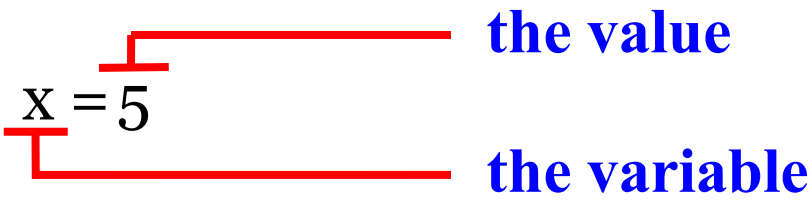
- Variables can be used in expressions
  - Evaluate to the value that is in the box
  - **Example**: x | 5 | **1 + x** evaluates to **6**
- Variables can change values
  - **Example**: x | ~~5~~ 1.5 | **1 + x** evaluates to **2.5**
  - Can even change the **type** of their value
  - Different from other languages (e.g. Java)

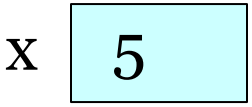
Naming Variables

- Python has strict rules of how to assign names
  - Names must only contain letters, numbers, _
  - They cannot start with a number
- **Examples**
  - e1 is a **valid** name
  - 1e2 is **not valid** (it is a **float**)
  - a_b is a **valid** name
  - a+b is **not valid** (it is + on two variables)

Variables and Assignment Statements

- Variables are created by **assignment statements**





- This is a **statement**, not an **expression**
  - **Expression**: Something Python turns into a value
  - **Statement**: Command for Python to do something
  - Difference is that has no value itself
- **Example**:

```
>>> x = 5
(NOTHING)
```

But can now use x as an expression

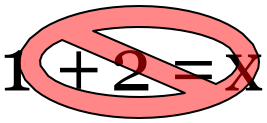
Variables Do Not Exist Until Made

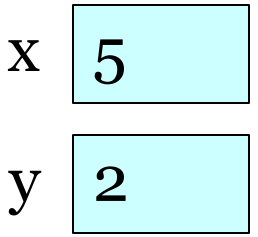
- Example:

```
>>> y
Error!
>>> y = 3
>>> y
3
```

- Changes our model of Python
  - Before we just typed in one line at a time
  - Now program is a **sequence** of lines

Assignments May Contain Expressions

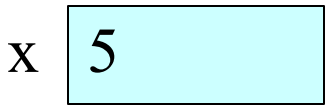
- **Example**: x = 1 + 2
  - Left of equals must always be variable: 1 + 2 = x 
  - Read assignment statements right-to-left!
  - Evaluate the expression on the right
  - Store the result in the variable on the left
- We can include variables in this expression
  - **Example**: x = y+2
  - **Example**: x = x+2



This is not circular!
Read right-to-left.

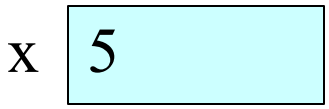
Execute the Statement: x = x + 2

- Draw variable x on piece of paper:



Execute the Statement: x = x + 2

- Draw variable x on piece of paper:



- Step 1: evaluate the expression x + 2
  - For x, use the value in variable x
  - Write the expression somewhere on your paper

Execute the Statement: x = x + 2

- Draw variable x on piece of paper:

- Step 1: evaluate the expression x + 2
  - For x, use the value in variable x
  - Write the expression somewhere on your paper
- Step 2: Store the value of the expression in x
  - Cross off the old value in the box
  - Write the new value in the box for x

Execute the Statement: x = x + 2

- Draw variable x on piece of paper:

- Step 1: evaluate the expression x + 2
  - For x, use the value in variable x
  - Write the expression somewhere on your paper
- Step 2: Store the value of the expression in x
  - Cross off the old value in the box
  - Write the new value in the box for x
- Check to see whether you did the same thing as your neighbor, discuss it if you did something different.

Which One is Closest to Your Answer?

A:

x 5 7

B:

x 5

x 7

C:

x 5

x 7

D:

¯_(ツ)_/¯

Which One is Closest to Your Answer?

A:

x 7

B:

x 5

x 7

C:

x 5

x 7

$x = x + 2$

Execute the Statement: x = 3.0 * x + 1.0

- You have this:

x ~~5~~ 7

Execute the Statement: x = 3.0 * x + 1.0

- You have this:

  x ~~5~~ 7

- Execute this command:
  - Step 1: **Evaluate** the expression 3.0 * x + 1.0
  - Step 2: **Store** its value in x

Execute the Statement: x = 3.0 * x + 1.0

- You have this:

  x ~~5~~ 7

- Execute this command:
  - Step 1: **Evaluate** the expression 3.0 * x + 1.0
  - Step 2: **Store** its value in x
- Check to see whether you did the same thing as your neighbor, discuss it if you did something different.

Which One is Closest to Your Answer?

A:

x ~~5~~ ~~7~~ 22.0

B:

x ~~5~~ 7

x 22.0

C:

x ~~5~~ ~~7~~

x 22.0

D:

¯_(ツ)_/¯

Which One is Closest to Your Answer?

A:

x 22.0

B:

x 7

x 22.0

C:

x

x 22.0

$$x = 3.0 * x + 1.0$$

Execute the Statement: x = 3.0 * x + 1.0

- You now have this:

  x ~~5~~ ~~7~~ 22.0

- The command:
  - Step 1: **Evaluate** the expression 3.0 * x + 1.0
  - Step 2: **Store** its value in x
- This is how you execute an assignment statement
  - Performing it is called **executing the command**
  - Command requires both **evaluate** AND **store** to be correct
  - Important *mental model* for understanding Python

Exercise: Understanding Assignment

- Add another variable, interestRate, to get this:

x ~~5~~ ~~7~~ 22.0 interestRate 4

- Execute this assignment:

```
interestRate = x / interestRate
```

- Check to see whether you did the same thing as your neighbor, discuss it if you did something different.

Which One is Closest to Your Answer?

A:

x 5 7 22.0 5.5

interestRate 4 5.5

B:

x 5 7 22.0

interestRate 4

interestRate 5.5

C:

x 5 7 22.0

interestRate 4 5.5

D:

x 5 7 22.0

interestRate 4 5

Which One is Closest to Your Answer?

A:

x ~~5~~ ~~7~~ 22~~.~~0 5.5

interestRate ~~4~~

B:

x ~~5~~ ~~7~~ 22.0

~~4~~

5.5

C:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~ 5.5

E:

¯_(ツ)_/¯

interestRate ~~4~~ 5

Which One is Closest to Your Answer?

interestRate = x/interestRate

B:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~

interestRate 5.5

C:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~ 5.5

D:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~ 5

Exercise: Understanding Assignment

- You now have this:

x ~~5~~ ~~7~~ 22.0 interestRate ~~4~~ 5.5

- Execute this assignment:

```
intrestRate = x + interestRate
```

- Check to see whether you did the same thing as your neighbor, discuss it if you did something different.

Which One is Closest to Your Answer?

A:

x 22.0

interestRate 27.5

B:

x 22.0

interestRate 5.5

intrestRate 27.5

C:

x 22.0 27.5

interestRate 5.5

D:

x 22.0

interestRate 5.5

intrestRate 27.5

Which One is Closest to Your Answer?

A:

x 22.0

interestRate

B:

x 22.0

estRate 5.5

27.5

C:

x 22.0

interestRate 5.5

interestRate

intrestRate 27.5

E:

¯_(ツ)_/¯

Which One is Closest to Your Answer?

A:

x 22.0

interestRate 27.5

B:

x 22.0

interestRate 5.5

intrestRate 27.5

```
intrestRate = x + interestRate
    ^
    e
```

Which One is Closest to Your Answer?

A:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~ ~~5.5~~ 27.5

B:

x ~~5~~ ~~7~~ 22.0

interestRate ~~4~~ 5.5

intrestRate 27.5

```
intrestRate = x + interestRate
    ^
    e
```

Spelling mistakes in Python are bad!!

Dynamic Typing

- Python is a **dynamically typed language**
  - Variables can hold values of any type
  - Variables can hold different types at different times
- The following is acceptable in Python:

```
>>> x = 1
>>> x = x / 2.0
```

  ← x contains an **int** value

  ← x now contains a **float** value

- Alternative is a **statically typed language**
  - Each variable restricted to values of just one type
  - This is true in Java , C, C++, etc.

Dynamic Typing

- Often want to track the type in a variable
  - What is the result of evaluating x / y?
  - Depends on whether x, y are **int** or **float** values
- Use expression type(<expression>) to get type
  - type(2) evaluates to <type 'int'>
  - type(x) evaluates to type of contents of x
- Can use in a boolean expression to test type
  - type('abc') == str evaluates to **True**